Python OO course

Tool used: spyder 4.1.5 (inside Anaconda nav)

Python version: 3.8

# Classes and instances

Procedural languages focus on functions, OO programming stresses on objects.

An object is a **collection of data** (variables) and **methods** (functions) that act on those data. Similarly a **class is a blueprint for that object**.

We can think of a class as a sketch (**prototype**) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. “House” is the object.

As many houses can be made from a house’s blueprint, **we can create many objects from a class**. An **object** is also called an **instance of a class** and the process of creating this object is called **instantiation**.

Defining a class:

Class myNewClass:

Pass

(the “pass” keyword is used because python expects you to type something there, remember to use correct indentation)

A class creates a **new local namespace** where all its attributes (data or functions) are defined.

**Special attributes**: they begin with **double underscores”\_\_”:**

* “\_\_doc\_\_”: gives us the **docstring** of the class (used to document a specific segment of code, describe WHAT it does, not HOW). You can print it out by “print(<classname>.\_\_doc\_\_)”
* “\_\_init\_\_”: gets called whenever a new object of that class is **instantiated** (=constructor) and is used to initialize all the variables.

Defining a class with init:

class room:

"this is my room docspace"

def \_\_init\_\_(self, name, size):

self.name = name

self.size = size

myroom=room("My own room",125)

print(myroom.name)

print(myroom.size)

print(myroom.\_\_doc\_\_)

The “self” keyword should be used (same as ‘this’ in other languages) to follow conventions. It refers to the object itself. **The \_\_init\_\_ takes at least 1 argument “self”**

During instantiation, the self does not have to be defined, it happens automatically, “myroom” will be passed in as “self”. f

# Methods and functions

**Definitions**:

* **Function**: block of code to carry out a specific task, with it’s own scope and called by a name. It can contain no or more args, on exit a function can or cannot return one or more values.
* **Method**: similar to a function except it is **associated with objects/classes**. It is implicitly used for an object for which it is called. The method is accessible to data that is contained within the class.

Example of a function:

def functionName( arg1, arg2,….):

   …….

   # Function\_body

   ……..

Example of a method:

class ClassName:

   def method\_name():

      …………..

      # Method\_body

      ………………

**Another type of function**:

* **Lambda** function (anonymous function):
  + Don’t have a body and are not required to call
  + Directly declared using the “lambda” keyword
  + Syntax :

lambda [arg1 [,arg2,.....argn]]:expression

**Normal function vs lambda function**:

# normal function linear expression

def lin(x):

return 3\*x + 2

print(lin(2))

# lambda function

f = lambda x: 3\*x + 2

print(f(2))

**Function arguments:**

* **Required** arg
  + Args that are passed in sequential order to a function
  + Number of args defined in a function should match with the function def

def addition(a, b):

sum = a+b

print("Sum after addition: ",sum)

addition(5, 6)

* **Keyword** arg
  + the use is done in a function call, the caller identifies the arg(s) by the arg name

def language(lname):

print(“We are learning a language called: ”, lname)

language(lname = “Python”)

* **Default** arg
  + function is called without any args, then it uses the default arg

def country(cName = “India”):

print(“Current country is:”, cName)

country(“New York”)

country(“London”)

country()

* **Variable-length** arg
  + if you want to process more args in a function than what you specified while defining a function. Inside the function you have to iterate over the tuple.
  + We need this because we can only define a functions once in Python, so we cannot define the same function with x args again with x+1 args etc

def add(\*num):

sum = 0

for n in num:

sum = n+sum

print(“Sum is:”, sum)

add(2, 5)

add(5, 3, 5)

add(8, 78, 90)

# Exceptions

Many people equate "exception" with "error," and this is particularly unsurprising given that **(a)** most exception classes have "**Error**" in their names, **(b)** most exceptions are indeed **indicative** of errors, and **(c)** an **unhandled** **exception** causes a program to exit. And you can talk about how an "**unhandled exception**" isn't the same as a **crash**, but those comments will fall on deaf ears when your manager is asking you why the program halted in production.  
  
It's typical for us to **trap** exceptions with "**try**" and "**except**".  But it's totally fine, and even good, to raise exceptions on occasion -- indicating where something unusual or problematic is going on. You **don't want to raise built-in Python exceptions**, both because they can be mistaken for something else and because it's a big advantage to have specific and clear exceptions in your code.  Sure, you could look through the message to decide what the problem is, but in my experience, you're likely to identify and fix a problem much faster based on a clear and unique exception, rather than by reading the message that comes alongside a generic one.

This is how you create your **own exceptions**:

class NotEnoughSpaceError(Exception):

pass

You define a class, make sure it **inherits** from "Exception", and give it an **empty body** (i.e., "pass").  Sure, you could add new attributes and methods, but why would you?  Exceptions have a standard API, and provide a particular set of data points about your program. Unless you have a really special kind of exception (e.g., the "re.error" exception class for regular expressions), no one is going to look beyond the exception class name and its message.

How can you raise this exception? Well, you're creating a new object of type NotEnoughSpaceError. So you'll say:

raise NotEnoughSpaceError(

f'{[one\_item.name](http://one_item.name/)} needs {one\_item.size}; only {self.available\_space - self.size()} available')

Take advantage of the f-string so that the message is as clear as can be.

(more info on f-strings -> <https://realpython.com/python-f-strings/> )

# Python namespaces

A namespace is a mapping from names to objects, with the property that there is zero relation between names in different namespaces. They’re usually implemented as Python dictionaries.

Python **classes** and **instances of classes** have their **own** **distinct** **namespaces**. When you try to access an attribute from an instance of a class, it **first** looks at its instance namespace. If it finds the attribute, it returns the associated value, if not, it then looks in the class namespace and returns the attribute (otherwise it presents an error).

The **instance** namespace takes **supremacy** over the class namespace. So **watch out** if you use the same attribute names in both namespaces.

classes can have an attribute too **outside of the instance** of the class.

when the **same attribute name** exists inside the \_\_init\_\_ section, Python will try to get that value (of the instance) and not the class-attribute. This has something to do with **namespaces**, here is an article that explained it to me:  <https://www.toptal.com/python/python-class-attributes-an-overly-thorough-guide>

Use these **class attributes** generally in the **following situations**:

* Storing **constants** that are used throughout the whole class
* Defining **static default values** (but beware, if default values can change with an instance of a class, you should define is in the instance of the class, not on the class itself)
* **Tracking data across all instances** of a given class (eg. calculating totals over all instances)
* **Performance**